A simple algorithm for computing the Lempel-Ziv factorization

Maxime Crochemore, Lucian Ilie, William F. Smyth

A simple algorithm for computing the Lempel–Ziv factorization

Maxime Crochemore[1, 2, *] Lucian Ilie[3, †, ‡] W. F. Smyth[4, 5, §]

[1] *Department of Computer Science, King's College London, London WC2R 2LS, UK*
[2] *Institut Gaspard-Monge, Université Paris-Est, F-77454 Marne-la-Vallée Cedex 2*
[3] *Department of Computer Science, University of Western Ontario, London, ON, N6A 5B7, Canada*
[4] *Department of Computing and Software, McMaster University, Hamilton, ON, L8S 4K1, Canada*
[5] *Digital Ecosystems & Business Intelligence Institute, Curtin University of Technology, Perth WA 6845, Australia*

Abstract

We give a space-efficient simple algorithm for computing the Lempel–Ziv factorization of a string. For a string of length n over an integer alphabet, it runs in $\mathcal{O}(n)$ time independently of alphabet size and uses $o(n)$ additional space.

1 Introduction

The *Lempel–Ziv factorization* of w [16] is the decomposition $w = u_0u_1 \cdots u_k$, where each u_i (except possibly u_k) is the longest prefix of $u_iu_{i+1} \cdots u_k$ that has another occurrence to the left in w or a single letter in case this prefix is empty. For example, the string abbaabbbaaabab has the Lempel–Ziv factorization a.b.b.a.abb.baa.ab.ab.

The Lempel–Ziv factorization is a basic and powerful technique for text compression [23]. Introduced to analyze the entropy of strings it has many variants used in gzip or PKzip software, and, more generally, in dictionary compression methods. The above factorization is specifically used in LZ77-based adaptive compression methods (see [21] or Section 2.5 in [22]).

The factorization plays an important role in String Algorithms. The intuitive reason is that when processing a string online, the work done on an element of the factorization can usually be skipped because already done on its previous occurrence. A typical application of this idea resides in algorithms to compute repetitions in strings, such as Kolpakov and Kucherov algorithm for reporting all maximal repetitions [15], and indeed it seems to be the only technique that leads to linear-time algorithms independently of the alphabet size (see [5]).

To compute the Lempel–Ziv factorization, some methods use suffix trees [20], others suffix automata [3], but these two data structures are not the most space-efficient. Recently

* Research supported in part by CNRS; e-mail: maxime.crochemore@kcl.ac.uk
† Research supported in part by NSERC; e-mail: ilie@csd.uwo.ca
‡ Corresponding author
§ Research supported in part by NSERC; e-mail: smyth@mcmaster.ca

algorithms have been proposed [1, 5, 2] that use suffix arrays, a more space-efficient structure. We improve here the simplest of those, that of [5], so that it uses $o(n)$ additional space as opposed to $\mathcal{O}(n)$ of [5].

2 Suffix arrays

We recall in this section briefly the notions of suffix array and longest common prefix. Consider a string $w = w[0\,..\,n-1]$ of length n over an integer alphabet A, that is, an integer interval of size no more than n^c, for some constant c. The suffix of w starting at position i is denoted by $\mathsf{suf}_i = w[i..n-1]$, for $0 \le i \le n-1$. The *suffix array* of w, [18], denoted SA, gives the suffixes of w sorted ascendingly in lexicographical order, that is, $\mathsf{suf}_{\mathsf{SA}[0]} < \mathsf{suf}_{\mathsf{SA}[1]} < \cdots < \mathsf{suf}_{\mathsf{SA}[n-1]}$. The suffix array of the string abbaabbbaaabab is shown in the second column of Fig. 1.

i	$w[i]$	$\mathsf{SA}[i]$	$\mathsf{LCP}[i]$	$\mathsf{suf}_{\mathsf{SA}[i]}$	$\mathsf{LPF}[i]$
0	a	8	0	aaabab	0
1	b	9	2	aabab	0
2	b	3	3	aabbbaaabab	1
3	a	12	1	ab	1
4	a	10	2	abab	3
5	b	0	2	abbaabbbaaabab	2
6	b	4	3	abbbaaabab	4
7	b	13	0	b	3
8	a	7	1	baaabab	2
9	a	2	3	baabbbaaabab	3
10	a	11	2	bab	2
11	b	6	1	bbaaabab	2
12	a	1	4	bbaabbbaaabab	2
13	b	5	2	bbbaaabab	1

Figure 1. The arrays SA, LCP, and LPF for the string $w =$ abbaabbbaaabab.

The suffix array of a string of length n over an integer alphabet can be computed in $\mathcal{O}(n)$ time by any of the algorithms in [10, 12, 13]; these algorithms are inspired by the $\mathcal{O}(n)$ suffix tree construction algorithm of [7].

Often the suffix array is used in combination with another array, the Longest Common Prefix (LCP) which gives the length of the longest common prefix between consecutive suffixes of SA, that is, $\mathsf{LCP}[i]$ is the length of the longest common prefix of $\mathsf{suf}_{\mathsf{SA}[i]}$ and $\mathsf{suf}_{\mathsf{SA}[i-1]}$; see the fourth column of Fig. 1 for an example.

Recall that [11] give simple linear-time algorithms to compute the LCP array; its space complexity is improved in [19].

We shall need also the Longest Previous Factor (LPF) array, defined as follows. For any position i in w, $\mathsf{LPF}[i]$ gives the length of the longest factor of w starting at position i that occurs previously in w. Formally, if $w[i]$ denotes the ith letter of w and $w[i\,..\,j]$ is the factor $w[i]w[i+1]\ldots w[j]$, then

$$\mathsf{LPF}[i] = \max\big(\{\ell \mid w[i..i+\ell-1] \text{ is a factor of } w[0..i+\ell-2]\} \cup \{0\}\big).$$

LPF was introduced in [5], but appears also as the λ array in [8]. In our example, the LPF array as defined above is given in column 6 of Fig. 1.

3 The algorithm

As already mentioned, we compute first the LPF array. The Lempel–Ziv factorization is then easily computed from LPF by the algorithm of Fig. 2, already proposed in [5]. For the example text $w =$ abbaabbbaaabab of Fig. 1, this algorithm outputs the sequence of starting positions of factors, $\mathsf{LZ} = [0, 1, 2, 3, 4, 7, 10, 12]$.

```
LEMPEL–ZIV_FACTORIZATION(LPF)
1.   LZ[0] ← 0; i ← 0
2.   while (LZ[i] < n) do
3.       LZ[i + 1] ← LZ[i] + max(1, LPF[LZ[i]])
4.       i ← i + 1
5.   return LZ
```

Figure 2. Algorithm for computing Lempel–Ziv factorization using LPF.

In order to explain the idea for the computation of LPF, it is helpful to arrange the SA and LCP arrays in a graph, as done in [5]. The vertices are labeled by the SA values and the edges by the LCP values. The vertices are arranged in left-to-right order corresponding to their order in SA and are placed at a height corresponding to their starting position in the string. In other words, if $\mathsf{SA}[i] = j$, then the vertex labeled j is plotted with abscissa i and ordinate j. Each edge between two vertices is labeled by the corresponding LCP value. An example if shown by the graph in Fig. 3(i) (solid edges only). Note that the graph is purely conceptual: no graph is constructed by the algorithm.

The value $\mathsf{LPF}[\mathsf{SA}[i]]$ can be immediately computed, that is, by local test only, in any of the following cases:

(i) $\mathsf{SA}[i-1] < \mathsf{SA}[i] > \mathsf{SA}[i+1]$, that is, for a "peak" in the graph. In this case $\mathsf{LPF}[\mathsf{SA}[i]] = \max(\mathsf{LCP}[i], \mathsf{LCP}[i+1])$. Referring to Fig. 3, for $i = 3$, the value $\mathsf{LPF}[\mathsf{SA}[3]] = \mathsf{LCP}[12]$ can be computed and it equals $\max(\mathsf{LCP}[3], \mathsf{LCP}[4]) = 2$ (maximum of the labels of the two adjacent edges). Since $\mathsf{SA}[3] = 12$, the vertex labeled 12 can then be removed from the graph. An edge between 3 and 10 is created, labeled by $\min(\mathsf{LCP}[3], \mathsf{LCP}[4]) = 1$ (minimum of the two labels).

(ii) $\mathsf{SA}[i-1] < \mathsf{SA}[i] < \mathsf{SA}[i+1]$ and $\mathsf{LCP}[i] \geq \mathsf{LCP}[i+1]$, that is, the SA-values are increasing but the LCP-values are decreasing. In this case nothing better (i.e., larger) than $\mathsf{LCP}[i]$ can be obtained from $\mathsf{LCP}[i+1]$ and so $\mathsf{LPF}[\mathsf{SA}[i]] = \mathsf{LCP}[i]$. For $i = 6$, the value $\mathsf{LPF}[\mathsf{SA}[6]] = \mathsf{LCP}[4]$ can be computed and it is equal to 3. As before, since $\mathsf{SA}[6] = 4$, the vertex 4 can be removed and the vertices 0 and 13 connected by an edge labeled $0 = \mathsf{LCP}[i+1]$.

(iii) $\mathsf{SA}[i-1] > \mathsf{SA}[i] > \mathsf{SA}[i+1]$ and $\mathsf{LCP}[i] \leq \mathsf{LCP}[i+1]$, that is, the SA-values are decreasing but the LCP-values are increasing. This is symmetric to (ii). However, since we consider the vertices from left to right, the case (i) will prevent the case (iii) from ever being used.

We consider then the vertices in the order given by SA and use the above cases (i)-(ii) any time we can. We maintain a stack with positions waiting to be processed. A fake

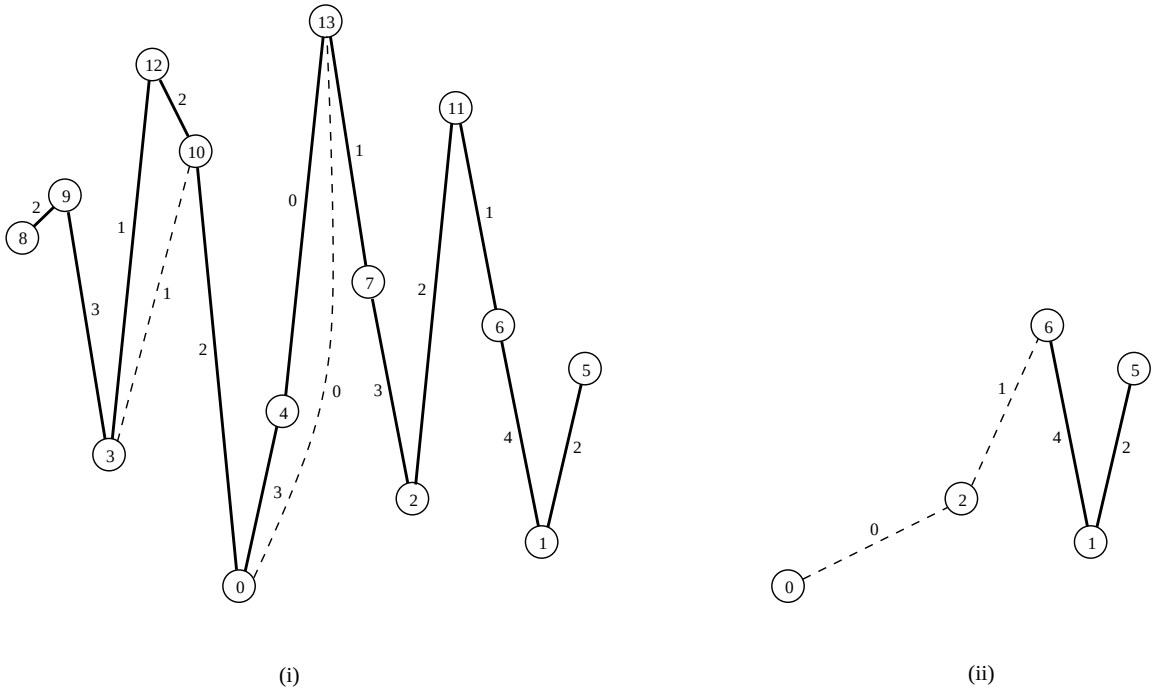

Figure 3. (i) Solid labeled edges form the graph representing SA and LCP for the text abbaabbbaaabab. (ii) The graph right before considering the vertex labeled 6.

position is added at the end of SA — that is, $\mathsf{SA}[n] = -1$, $\mathsf{LCP}[n] = 0$ — to make sure that all positions are considered uniformly. The algorithm for computing LPF is given in Fig. 4. Fig. 3(ii) shows the modified graph right before the vertex labeled 6 is considered in step 3 ($i = 11$).

The correctness of the algorithm follows from the above discussion. It runs in $\mathcal{O}(n)$ time because each position $i, 0 \leq i \leq n-1$, is pushed at most once onto the stack.

The difference between this algorithm and the one of [5] starts with the fact that the latter uses only the case (i) above. The use of case (ii) reduces drastically the amount of additional space needed, from $\mathcal{O}(n)$ to $o(n)$, as proved by a combinatorial argument that we present below.

Consider the space used by the stack. As noted above, the positions in the stack at any moment are in increasing order (from bottom to top), and both their SA- and LCP-values are strictly increasing as well. That is, if the content of the stack is (from bottom to top) $i_1 < i_2 < \cdots < i_k$, then

$$\mathsf{SA}[i_1] < \mathsf{SA}[i_2] < \cdots \mathsf{SA}[i_k] \quad \text{and} \quad \mathsf{LCP}[i_1] < \mathsf{LCP}[i_2] < \cdots \mathsf{LCP}[i_k].$$

According to our algorithm, $\mathsf{LCP}[i_j]$ contains at this moment the length of the longest common prefix of $\mathsf{suf}_{i_{j-1}}$ and suf_{i_j}.

We now show that $i_{j+2} - i_j \geq \mathsf{LCP}[i_{j+1}]$ using string-combinatorial arguments; see, e.g., [17]. Assume the opposite inequality holds. Then the factor $w[i_j \,..\, i_{j+1} + \mathsf{LCP}[i_{j+1}] - 1]$ has period $i_{j+1} - i_j$ (due to overlap of the identical factors $w[i_j \,..\, i_j + \mathsf{LCP}[i_{j+1}] - 1]$ and $w[i_{j+1} \,..\, i_{j+1} + \mathsf{LCP}[i_{j+1}] - 1]$). Since $\mathsf{LCP}[i_{j+2}] > \mathsf{LCP}[i_{j+1}]$ and $w[i_{j+2} \,..\, i_{j+2} + \mathsf{LCP}[i_{j+1}] - 1]$ overlaps $w[i_{j+1} \,..\, i_{j+1} + \mathsf{LCP}[i_{j+1}] - 1]$ by at least $i_{j+1} - i_j$ positions, the primitive roots of the two periods must synchronize. Hence, the period $i_{j+1} - i_j$ continues past the position

```
COMPUTE_LPF(SA, LCP)
 1.  SA[n] ← −1; LCP[n] ← 0
 2.  PUSH(0, 𝒮)
 3.  for i from 1 to n do
 4.      while ((𝒮 ≠ ∅) and
 5.             ((SA[i] < SA[TOP(𝒮)]) or
 6.             ((SA[i] > SA[TOP(𝒮)]) and (LCP[i] ≤ LCP[TOP(𝒮)])))) do
 7.          if (SA[i] < SA[TOP(𝒮)]) then
 8.              LPF[SA[TOP(𝒮)]] ← max(LCP[TOP(𝒮)], LCP[i])
 9.              LCP[i] ← min(LCP[TOP(𝒮)], LCP[i])
10.          else
11.              LPF[SA[TOP(𝒮)]] ← LCP[TOP(𝒮)]
12.          POP(𝒮)
13.      if (i < n) then PUSH(i, 𝒮)
14.  return LPF
```

Figure 4. Algorithm for computing LPF.

$i_{j+1} + \mathsf{LCP}[i_{j+1}] - 1$ in w; that is, $\mathsf{LCP}[i_{j+1}]$ is strictly shorter than the actual length of the longest common prefix of suf_{i_j} and $\mathsf{suf}_{i_{j+1}}$, a contradiction.

Since $i_{j+2} - i_j \geq \mathsf{LCP}[i_{j+1}]$ and $\mathsf{LCP}[i_{j+1}] > \mathsf{LCP}[i_j]$, we easily obtain that $k = \mathcal{O}(\sqrt{n})$ and thus the maximum size of the stack is $o(n)$.

It is interesting to note that the upper bound we just proved on the maximum stack size is asymptotically optimal. For the strings $w = \mathsf{abab}^2\mathsf{ab}^3 \ldots \mathsf{ab}^\ell$, the stack needs to store simultaneously ℓ elements, $0, 2, 5, 9, \ldots, \frac{\ell(\ell+1)}{2} - 1$; that is, it requires $\Theta(\sqrt{n})$ space.

We have proved

Theorem 1 *The* LPF *array and the Lempel–Ziv factorization of a string of length n over an integer alphabet can be computed, using* SA *and* LCP*, in $\mathcal{O}(n)$ time independently of alphabet size and $o(n)$ additional space.*

4 Conclusion

We discuss briefly a number of variations of our COMPUTE_LPF algorithm that are possible with small changes.

First, all features of the algorithm of [5] are preserved. One of them is that we can keep LCP unchanged simply by storing $(i, \mathsf{LCP}[i])$ pairs on the stack. Another one, the algorithm of [5] computes also the PrevOcc array, which gives a previous position where a factor of length $\mathsf{LPF}[i]$ starts. The present algorithm can compute PrevOcc in the same way.

Second, the LPF keyed to SA instead of w can be computed, that is, if we denote the new array by LPF′, then $\mathsf{LPF}'[i] = j$ iff $\mathsf{LPF}[\mathsf{SA}[i]] = j$. We need only change $\mathsf{LPF}[\mathsf{SA}[\text{TOP}(\mathscr{S})]]$ to $\mathsf{LPF}[\text{TOP}(\mathscr{S})]$ in steps 8 and 11.

This may be useful, for instance, if we want to save more space by computing LPF in place of LCP. (The LCP array is, by definition, keyed to SA.) In such a case, we would replace LPF by LCP and remove the steps 10 and 11.

Note that, for our initial purpose, computation of the Lempel–Ziv factorization, we would have then to compute LPF keyed to w, which can be done easily in place (using constant additional space).

References

[1] M.I. Abouelhoda, S. Kurtz, and E. Ohlenbusch, Replacing suffix trees with enhanced suffix arrays, *J. Discrete Algorithms* **2** (2004) 53 – 86.

[2] G. Chen, S.J. Puglisi, and W.F. Smyth, Fast and practical algorithms for computing all runs in a string, *Proc. of CPM'07*, Lecture Notes in Comput. Sci. **4580**, Springer, Berlin, 2007, 307 – 315.

[3] M. Crochemore, Transducers and repetitions, *Theoret. Comput. Sci.* **45**(1) (1986) 63 – 86.

[4] M. Crochemore, C. Hancart, and T. Lecroq, *Algorithms on Strings*, Cambridge Univ. Press, 2007.

[5] M. Crochemore and L. Ilie, Computing longest previous factor in linear time and applications, *Inform. Proc. Lett.*, to appear.

[6] J.-P. Duval, R. Kolpakov, G. Kucherov, T. Lecroq, and A. Lefebvre, Linear-Time Computation of Local Periods, *Theoret. Comput. Sci.* **326**(1-3) (2004) 229 – 240.

[7] M. Farach, Optimal suffix tree construction with large alphabets, in *Proc. of FOCS'97*, IEEE Computer Society Press, 1997, 137 - 143.

[8] F. Franek, J. Holub, W. F. Smyth and X. Xiao, Computing quasi suffix arrays, *J. Automata, Languages and Combinatorics* **8**(4) (2003) 593–606.

[9] D. Gusfield and J. Stoye, Linear Time Algorithms for Finding and Representing all the Tandem Repeats in a String, *J. Comput. Syst. Sci.* **69**(4) (2004) 525 – 546.

[10] J. Kärkkäinen and P. Sanders, Simple linear work suffix array construction, in *Proc. of ICALP'03*, Lecture Notes in Comput. Sci. **2719**, Springer-Verlag, Berlin, Heidelberg, 2003, 943 - 955.

[11] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park, Linear-time longest-common-prefix computation in suffix arrays and its applications, *Proc. of CPM'01*, Lecture Notes in Comput. Sci. **2089**, Springer-Verlag, Berlin, 2001, 181 - 192.

[12] D.K. Kim, J.S. Sim, H. Park, and K. Park, Constructing suffix arrays in linear time. *J. Discrete Algorithms* **3**(2-4) (2005) 126 - 142.

[13] P. Ko and S. Aluru, Space efficient linear time construction of suffix arrays, *J. Discrete Algorithms* **3**(2-4) (2005) 143 - 156.

[14] R. Kolpakov and G. Kucherov, Finding maximal repetitions in a word in linear time, in: *Proceedings of the 40th FOCS*, IEEE Computer Society Press, New York, 1999, 596–604.

[15] R. Kolpakov and G. Kucherov, On maximal repetitions in words, *J. Discrete Algorithms* **1**(1) (2000) 159 – 186.

[16] A. Lempel and J. Ziv, On the complexity of finite sequences, *IEEE Trans. Inform. Theory* **92**(1) (1976) 75 – 81.

[17] M. Lothaire, *Algebraic Combinatorics on Words*, Cambridge Univ. Press, 2002.

[18] U. Manber and G. Myers. Suffix arrays: a new method for on-line search, *SIAM J. Comput.* **22**(5) (1993) 935 – 948.

[19] G. Manzini, Two space-saving tricks for linear-time LCP computation, in T. Hagerup & J. Katajainen (eds.) *Proc. SWAT 2004*, Lecture Notes in Comput. Sci. **3111** (2004) 372–383.

[20] M. Rodeh, V.R. Pratt, and S. Even, Linear Algorithm for Data Compression via String Matching, *J. ACM* **28**(1) (1981) 16 – 24.

[21] J. Storer and T. Szymanski, Data compression via textual substitution, *J. ACM* **29**(4) (1982) 928 – 951.

[22] I.H. Witten, A. Moffat, and T.C. Bell, *Managing Gigabytes*, Van Nostrand Reinhold, New York, 1994.

[23] J. Ziv and A. Lempel, A universal algorithm for sequential data compression, *IEEE Trans. Inform. Theory* **23**(3) (1977) 337 – 343.